Programming Fundamentals 3 **2021**

Instructor: Prof. Walter Binder **TA:** Filippo Schiavio, Eduardo Rosales, Matteo Basso

---

Assignment 3

Due date: 21 December 2021, 08:00 a.m.

For questions regarding this assignment, please contact Filippo Schiavio via email: `filippo.schiavio@usi.ch`.

This assignment contributes 10% to the overall grade. Please follow strictly the submission instructions written at the end of the assignment.

Environment Setup

The exercises presented in this assignment are available as a single maven project in the attached folder: **/Ex**. Before proceeding, you are advised to properly set the `JAVA_HOME` variable and you must install maven following the official installation guide.

Compile and Test the Projects

To compile the projects, you should use the command: `mvn compile`. For instance:

1. Open your terminal
2. Navigate to the root folder of the project, **/Ex**
3. Compile the source files:
   ```
   mvn compile
   ```
4. To test the project please follow the instructions provided for each exercise

If you need further details or you encounter errors, please refer to the Java Tools lecture recordings that includes a detailed explanation of maven and a step by step guide to install and run it, including running tests.

Exercise 1 - Simulating Kids Activities - CyclicBarrier (2 points)

You are organizing activities to be performed by kids during a party. Activities are predefined and cannot change once the party starts. Similarly, the set of kids performing these activities is fixed (i.e., a kid cannot join the party once it has started and a kid cannot leave the party before it is over).
The activities during the party must be performed by all kids in consecutive order, i.e., once a kid finishes an activity, he/she must wait for all other kids to finish the current activity before performing the next one. There is no time limit defining how long a kid performs an activity. Also, when all kids finish an activity, they must tidy up the tools used for the current activity before performing the next one.

All the Java files for this exercise are in the package `ex1`. This exercise requires to model each kid as a Java Thread instance. You should use the class `Kid` which is defined below and is included in the package **ex1** in project folder **Ex/**. As parameters, the constructor of `Kid` expects an array of `Activity`, representing the activities to be performed, and an instance of `java.util.concurrent.CyclicBarrier`. You should properly instantiate `CyclicBarrier` in the constructor of `Ex1Simulator` and implement the method `Kid.waitForOtherKids()`, which should take care of waiting for other threads. You should only use the Java class `CyclicBarrier` as synchronization mechanism to wait for other threads. The cleaning phase should be implemented using the provided class `ActivityCleaner`, a `Runnable` which should be executed once for each activity, after all kids have performed it.

You are not allowed to directly invoke the method `ActivityCleaner.run()` in your code, instead, you should instantiate the `CyclicBarrier` so that the method `ActivityCleaner.run()` is automatically executed by the last thread which reaches the barrier.

We do not provide JUnit tests for this exercise; however, you can check your solution by running the provided simulation with the commands:

```
mvn compile
mvn -q exec:java@ex1
```

The following is an example of the expected output on a correct solution (truncated after the first two activities). Note that the sentence: *"Cleaning done. The current activity has been completed."* must appear after all kids performed an activity.

```
Kid #2 performed Activity 0
Kid #4 performed Activity 0
Kid #1 performed Activity 0
Kid #3 performed Activity 0
Kid #0 performed Activity 0
Cleaning done. The current activity has been completed.
Kid #1 performed Activity 1
Kid #0 performed Activity 1
Kid #2 performed Activity 1
Kid #4 performed Activity 1
Kid #3 performed Activity 1
Cleaning done. The current activity has been completed.
....
```

Provided Java classes:

```java
public final class Activity {

  private final String name;
  private final Random random = new Random();

  public Activity(String name) {
    this.name = name;
  }

  public void perform() throws InterruptedException {
    Thread.sleep(random.nextInt(10)); // Simulate activity time
    System.out.println(Thread.currentThread().getName() + " performed " + name);
  }

  public String getName() {
    return name;
  }
}


public final class ActivityCleaner implements Runnable {
  @Override
  public void run() {
    System.out.println("Cleaning done. The current activity has been completed.");
  }
}

public final class Kid implements Runnable {

  private final Activity[] activities;
  private final CyclicBarrier barrier;

  public Kid(Activity[] activities, CyclicBarrier barrier) {
    this.activities = activities;
    this.barrier = barrier;
  }

  @Override
  public void run() {
    try {
      for (Activity activity : activities) {
        activity.perform();
        waitForOtherKids();
      }
    } catch (Exception e) {
      System.out.println("Exception occurred: " + e.getMessage());
      System.exit(1);
    }
  }

  private void waitForOtherKids() throws Exception {
    // TODO use the barrier to wait for all the other kids
  }
}
```

```java
public final class Ex1Simulator {

  public static void main(String[] args) {
    new Ex1Simulator().run();
  }

  private static final int N_KIDS = 5;
  private static final int N_ACTIVITIES = 10;

  public void run() {
    Activity[] activities = new Activity[N_ACTIVITIES];
    for (int i = 0; i < activities.length; i++) {
      activities[i] = new Activity("Activity " + i);
    }
    // TODO instantiate a CyclicBarrier
    CyclicBarrier barrier = null;

    for (int i = 0; i < N_KIDS; i++) {
      new Thread(new Kid(activities, barrier), "Kid #" + i).start();
    }
  }
}
```

Exercise 2 - Voting your Preferred Activity - Concurrency (2 points)

In the context of the previous exercise, the organizers would like to collect feedback from the kids about the performed activities. The organizers attempted to implement a voting system which should allow kids to concurrently vote for their preferred activity. Unfortunately, the testing of the system indicates that it is wrongly implemented.
All the Java files for this exercise are in the package `ex2`. The voting system implementation can be found in the class `VoteCounter`, and it is shown below.

```java
public final class VoteCounter {
  private final Map<String, Integer> votes = new HashMap<>();

  public void vote(String activity) {
    Integer currentCount = votes.get(activity);
    if(currentCount == null) {
      currentCount = 0;
    }
    votes.put(activity, currentCount + 1);
  }

  public int getVoteCount(String activity) {
    Integer count = votes.get(activity);
    return count == null ? 0 : count;
  }
}
```

For simplicity, an activity is represented by a string, i.e., the activity name.
Note that method `VoteCounter.vote(..)` is not thread-safe. Races can occur while updating the vote count for a given option and wrong vote counts can be reported. You should fix this issue and make method `VoteCounter.vote(..)` thread-safe.

Although the issue could be trivially fixed by making the methods `VoteCounter.vote(..)` and `VoteCounter.getVoteCount(..)` atomic (e.g., marking them as `synchronized`) you are not allowed to directly use any form of locking to solve this exercise. Instead, you should use a `ConcurrentHashMap`[1]`<String, AtomicInteger`[2]`>` as a map implementation for storing activity-vote mappings.
Then, you should modify methods `vote(..)` and `getVoteCount(..)` for dealing with instances of `AtomicInteger` instead of `Integer`. Note that you are not allowed to change the signature of `getVoteCount(..)`, i.e., it must return an `int` and not an `AtomicInteger`.

Changing the map implementation is not enough to guarantee atomicity in the compound action. Since the method `vote(..)` needs to put an entry in the map if the key is not already mapped to a value, you should call a single method on the map which ensures atomicity on this action.

Hint: The class `ConcurrentHashMap` offer multiple ways to insert a new entry in the map only if the key is not already mapped to a value or retrieving the mapped value otherwise in a single atomic method. Those methods are suitable for solving this exercise.

[1]java.util.concurrent.ConcurrentHashMap
[2]java.util.concurrent.atomic.AtomicInteger

Note: We provide a JUnit test for this exercise, which you can run with the commands:

```
mvn compile
mvn test -Dtest=Ex2Test
```

However, remember that races are non-deterministic such that you should run the tests multiple times. A single failure indicates a wrong implementation. Please also note that tests are useful only to check whether errors occur. They do not ensure that an exercise was solved correctly.
Other provided classes:

```java
public final class Ex2Simulator {

  public static void main(String[] args) throws InterruptedException {
    new Ex2Simulator().run();
  }

  public static final int N_KIDS = 1000;
  public static final int N_ACTIVITIES = 5;
  public static final String[] ACTIVITIES = new String[N_ACTIVITIES];
  static {
    for (int i = 0; i < N_ACTIVITIES; i++) {
      ACTIVITIES[i] = "Activity " + i;
    }
  }

  public VoteCounter run() throws InterruptedException {
    VoteCounter voteCounter = new VoteCounter();

    Thread[] threads = new Thread[N_KIDS];
    for (int i = 0; i < N_KIDS; i++) {
      Voter kid = new Voter(ACTIVITIES, voteCounter);
      threads[i] = new Thread(kid, "Kid #" + i);
    }
    for (int i = 0; i < N_KIDS; i++) {
      threads[i].start();
    }
    for (int i = 0; i < N_KIDS; i++) {
      threads[i].join();
    }

    return voteCounter;
  }
}


public final class Voter implements Runnable {

  private final String[] activities;
  private final VoteCounter voteCounter;
  private final Random random = new Random();

  public Voter(String[] activities, VoteCounter voteCounter) {
    this.activities = activities;
    this.voteCounter = voteCounter;
  }

  @Override
  public void run() {
    voteCounter.vote(activities[random.nextInt(activities.length)]);
  }
}
```

Exercise 3 - Perfect Square Parallel Counter - Thread Pools (6 points)

Consider the class `SequentialPerfectSquareCounter`, a Java implementation of a sequential algorithm to count the occurrences of perfect square numbers (i.e., an integer that is the square of an integer) within an integer array. The method `int countPrimes(int[] nums)`, defined in the interface `PerfectSquareCounter`, receives an integer array and returns the number of perfect squares in it.

```java
public final class PerfectSquare {
  public static boolean isPerfectSquare(int n) {
    if(n < 0) {
      return false;
    }
    int sqrt = (int) Math.round(Math.sqrt(n));
    return sqrt * sqrt == n;
  }
}

public interface PerfectSquareCounter {
  int countPerfectSquares(int[] nums) throws Exception;
}

public final class SequentialPerfectSquareCounter implements PerfectSquareCounter {

  public static int countPerfectSquares(int[] nums, int from, int to) {
    int count = 0;
    for(int i = from; i < to; i++) {
      if(PerfectSquare.isPerfectSquare(nums[i])) {
        count++;
      }
    }
    return count;
  }

  @Override
  public int countPerfectSquares(int[] nums) {
    return countPerfectSquares(nums, 0, nums.length);
  }
}
```

For this exercise you have to implement two parallel algorithms for counting the number of perfect squares in a given integer array. All the Java files for this exercise are in the package `ex3`.

3.1 Perfect Square Parallel Counter - FixedThreadPool (3 points)

Here, you should implement a parallel perfect square counter using a Java thread pool with a fixed number of threads. Consider the provided template of the class `ThreadPoolPerfectSquareCounter` defined below.

```java
public final class ThreadPoolPerfectSquareCounter implements PerfectSquareCounter {
  private static final int SEQUENTIAL_THRESHOLD = 1000;

  @Override
  public int countPerfectSquares(int[] nums) throws Exception {
    List<Callable<Integer>> tasks = new LinkedList<>();
    int from = 0, to = SEQUENTIAL_THRESHOLD;
    while(to < nums.length) {
      tasks.add(new CountPerfectSquarePortion(nums, from, to));
      from = to;
      to += SEQUENTIAL_THRESHOLD;
    }
    tasks.add(new CountPerfectSquarePortion(nums, from, nums.length));

    // TODO 1) instantiate an ExecutorService with a fixed number of threads
    //  equals to the number of available processors in the system
    ExecutorService executor = null;

    // TODO 2) execute all tasks with the executor and get a List<Future<Integer>>
    List<Future<Integer>> futures = null;

    // TODO 3) shutdown the threadpool and wait for its termination

    // TODO 4) add to count all integer values in the List<Future<Integer>>
    int count = 0;
    return count;
  }

  private static class CountPerfectSquarePortion implements Callable<Integer> {
    private final int[] nums;
    private final int low, high;

    private CountPerfectSquarePortion(int[] nums, int low, int high) {
      this.nums = nums; this.low = low; this.high = high;
    }

    @Override
    public Integer call() {
      return SequentialPerfectSquareCounter
          .countPerfectSquares(nums, low, high);
    }
  }
}
```

We provide the task class `CountPerfectSquarePortion` as well as the code that initializes all the tasks needed to solve the problem, i.e., one task for each portion of the array which sequentially counts the number of perfect squares in that portion. You should implement all the TODOs listed in the code, using a thread pool for executing the given tasks. In particular, you should take care of instantiating the thread pool, use it to execute all the provided tasks, shutdown the thread pool, wait for its termination, and return the sum of all values obtained executing the tasks.

Note: We provide a JUnit test for this exercise, which you can run with the commands:

```
mvn compile
mvn -Dtest=Ex3Test#testThreadPool test
```

3.2 Perfect Square Parallel Counter - ForkJoinPool (3 points)

Here, you should implement a parallel perfect square counter using a Java fork-join pool by decomposing the problem recursively. Consider the provided template of the class `ForkJoinPerfectSquareCounter` defined below.

```java
public final class ForkJoinPerfectSquareCounter implements PerfectSquareCounter {
  private static final int SEQUENTIAL_THRESHOLD = 1000;

  @Override
  public int countPerfectSquares(int[] nums) {
    CountPerfectSquarePortion task =
            new CountPerfectSquarePortion(nums, 0, nums.length);

    // TODO 1) execute the CountPerfectSquare task on
    //  the common ForkJoinPool and return the result
    return 0;
  }

  private static class CountPerfectSquarePortion extends RecursiveTask<Integer> {
    private final int low, high;
    private final int[] nums;

    public CountPerfectSquarePortion(int[] nums, int low, int high) {
      this.nums = nums; this.low = low; this.high = high;
    }

    @Override
    protected Integer compute() {
      int size = high - low;
      if (size <= SEQUENTIAL_THRESHOLD) {
        return SequentialPerfectSquareCounter
                .countPerfectSquares(nums, low, high);
      }
      else {
        int mid = low + size / 2;
        // TODO 2) Complete method compute by creating two
        //  CountPerfectSquare recursive tasks for computing
        //  the count of the first and second half of the input array;
        //  execute them in parallel and return the sum of their result
        return 0;
      }
    }
  }
}
```

Similarly to the exercise 3.1, we provide the task class `CountPerfectSquarePortion` which counts the number of perfect squares in a portion of the input array, if the size of that portion is below the given threshold `SEQUENTIAL_THRESHOLD`. However, this time you have to recursively create the tasks in the method `CountPerfectSquarePortion.compute()`. As described in the TODO, you should create two recursive tasks, which will compute the number of perfect squares in the first and second half of the current portion of the array. Then, you should also complete the implementation of the method `ForkJoinPerfectSquareCounter.countPerfectSquares` as described in the TODO. In particular, we provide the instantiation of the first task and you should implement the execution of the task in the common fork-join pool.

Note: We provide a JUnit test for this exercise, which you can run with the commands:

```
mvn compile
mvn -Dtest=Ex3Test#testForkJoinPool test
```

SUBMISSION INSTRUCTIONS

(please notice that the following instructions are mandatory.
Submissions that do not comply, will not be graded):

- Assignments are individual work. Duplicates are not graded.
- Assignments must be submitted via **iCorsi**. Do not send anything by email.
- Solutions to theoretical questions must be presented inside a (single) PDF or plain text file named:
  assignment<AssignmentNumber>.<Surname>.<Name><.pdf|.txt>
  In that document you must clearly specify your name and the question you are addressing. Please do not submit scanned documents using handwritten text.
- In addition, you must attach the whole maven project that you have modified to solve the assignment. Submissions without Java source code or with Java source code that does not compile will not be graded.
- All the produced files must be collected into a single archive file (.zip or .tar) named:
  assignment<AssignmentNumber>.<Surname>.<Name><.zip|.tar.gz>